JÜLICH
FORSCHUNGSZENTRUM

Parallel I/O and Portable Data Formats

I/O strategies

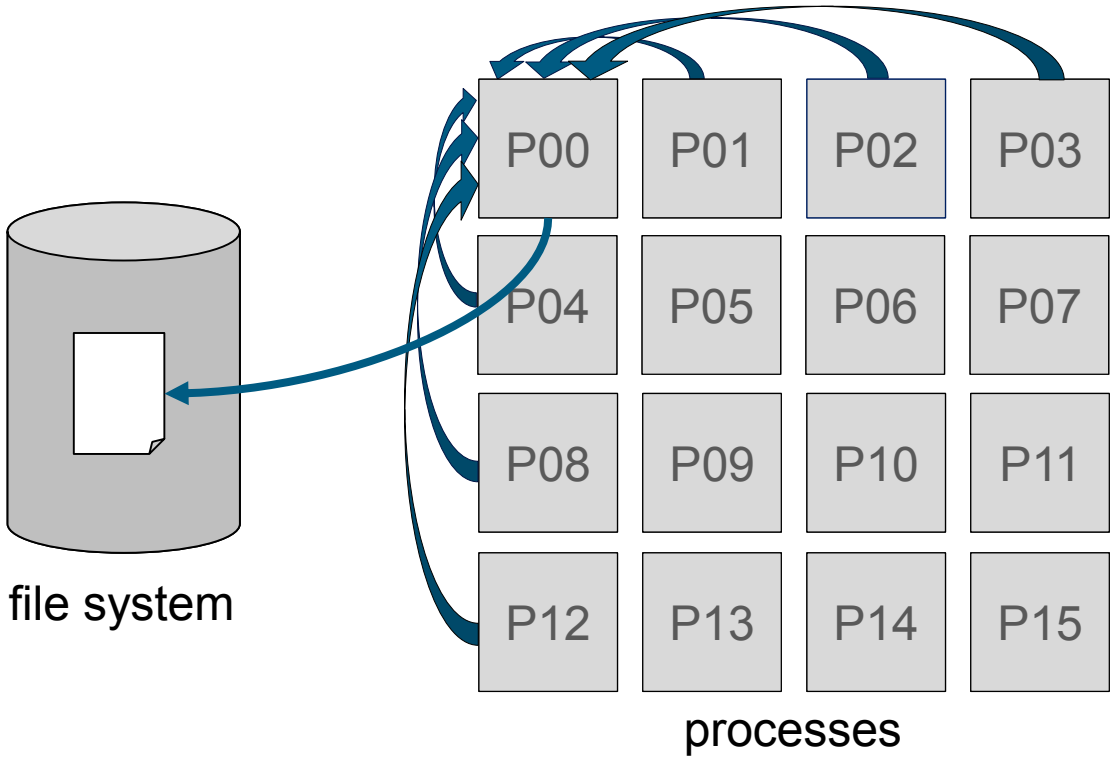
16 March 2015 | Sebastian Lührs

Outline

- Common I/O strategies
  - Pitfalls
- I/O Workflow
  - Pitfalls
- Parallel I/O Software Stack
- Course exercise description
  - General exercise workflow
  - Mandelbrot set description
  - Exercise API

Common I/O strategies

One process performs I/O



Common I/O strategies

One process performs I/O

+ Simple to implement

- I/O bandwidth is limited to the rate of this single process

- Additional communication might be necessary

- Other processes may idle and waste computing resources during I/O time

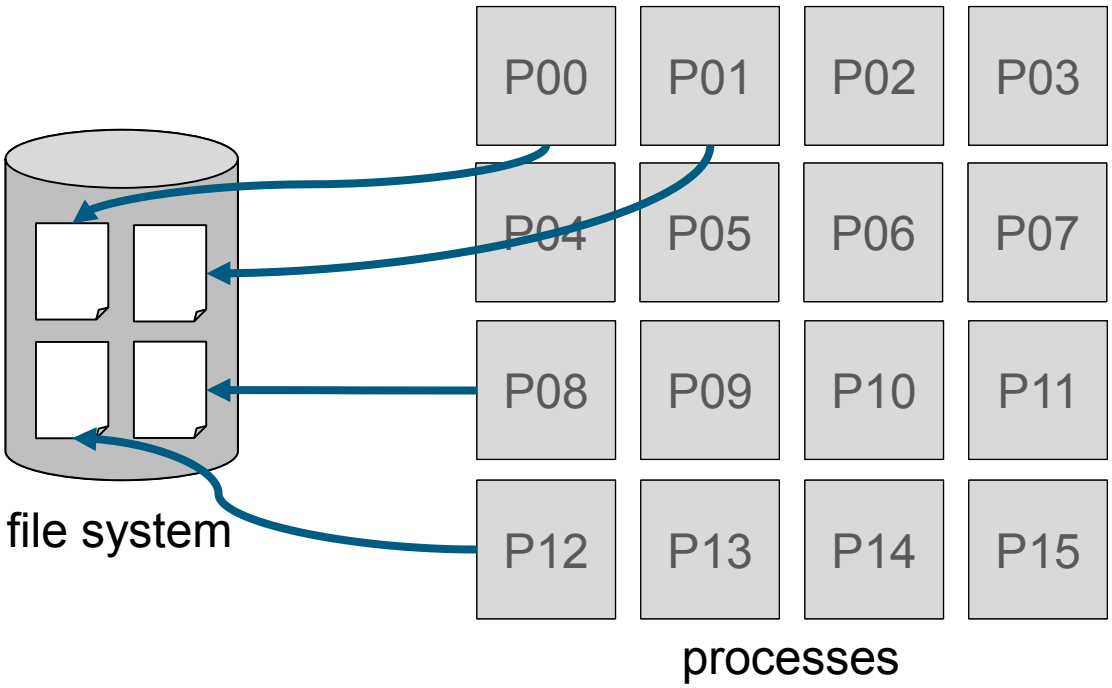
Common I/O strategies

Pitfall 1 – Frequent flushing on small blocks

- Modern file systems in HPC have **large file system blocks** (e.g. 4MB)
- A flush on a file handle forces the file system to perform all pending write operations
- If application writes in small data blocks, the same file system block it has to be **read and written multiple times**
- Performance degradation due to the inability to combine several write calls

Common I/O strategies

Each process writes to its own file (task-local files)



Common I/O strategies

Each process writes to its own file (task-local files)

+ Simple to implement

+ No coordination between processes needed

+ No false sharing of file system blocks

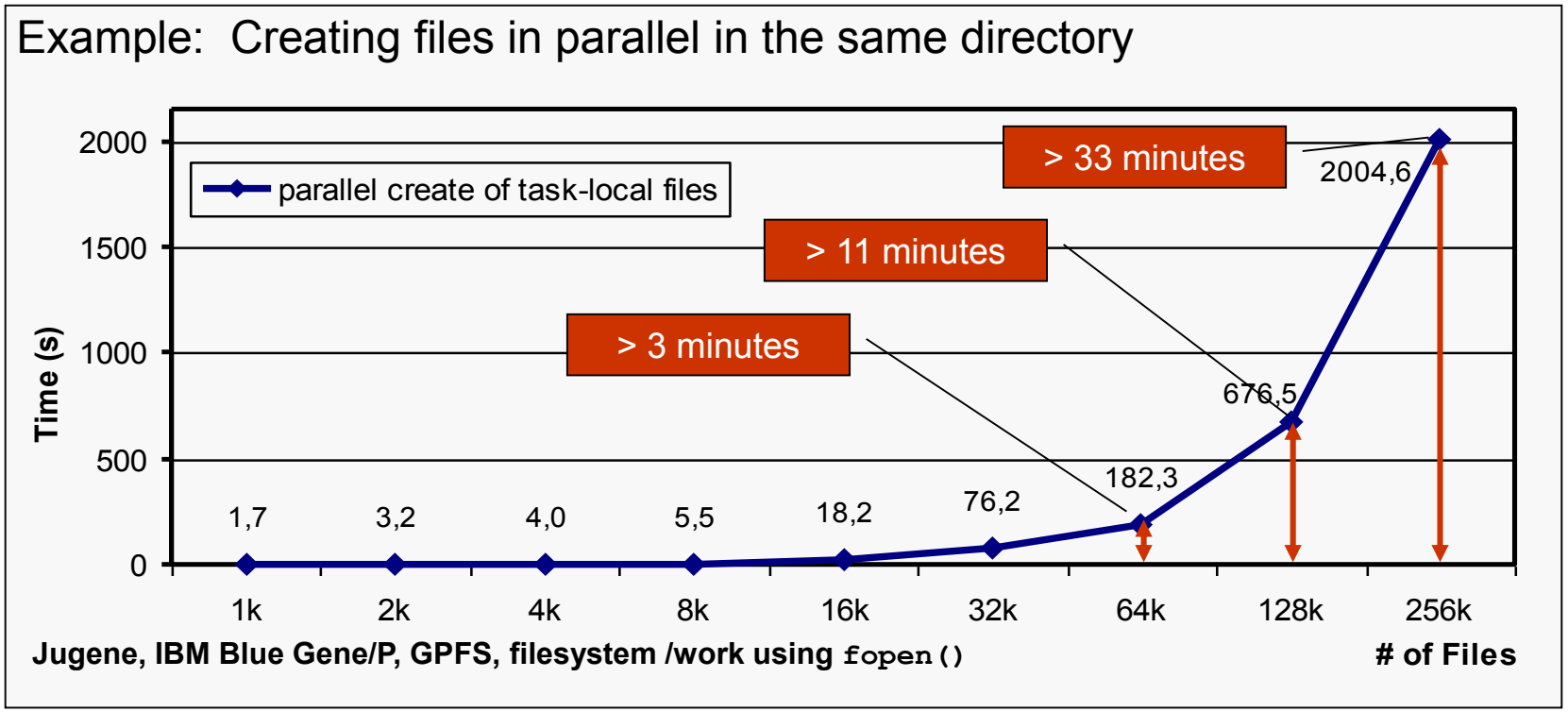
- Number of files quickly becomes unmanageable

- Files often need to be merged to create a canonical dataset

- File system might serialize meta data modification

Common I/O strategies

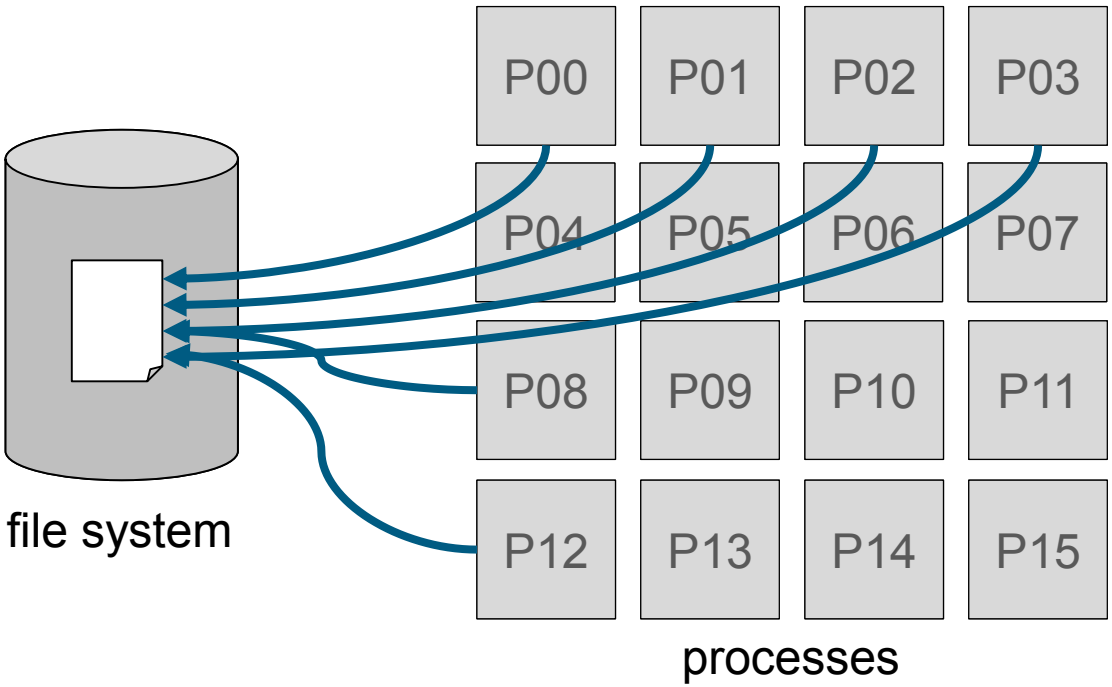
Pitfall 2 – Serialization of meta data modification



The creation of 256.000 files costs 142.500 core hours!

Common I/O strategies

All or several processes write to one file



Common I/O strategies

All or several processes write to one file

+ Number of files is independent of number of processes

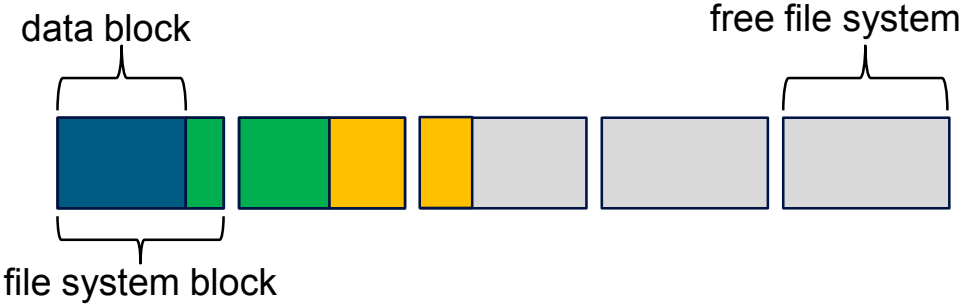
+ File can be in canonical representation (no post-processing)

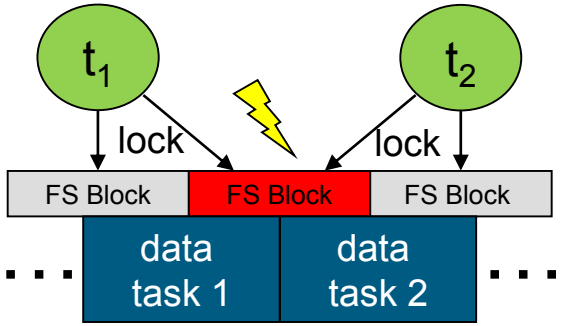
- Uncoordinated client requests might induce time penalties

- File layout may induce false sharing of file system blocks

Common I/O strategies

Pitfall 3 – False sharing of file system blocks



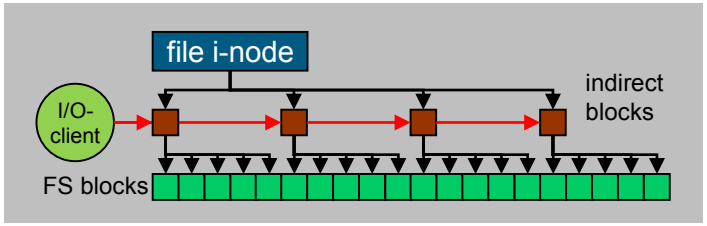


- Data blocks of individual processes **do not fill up a complete file system block**
- Several processes **share a file system block**
- Exclusive access (e.g. write) must be **serialized**
- The more processes have to synchronize the more waiting time will propagate

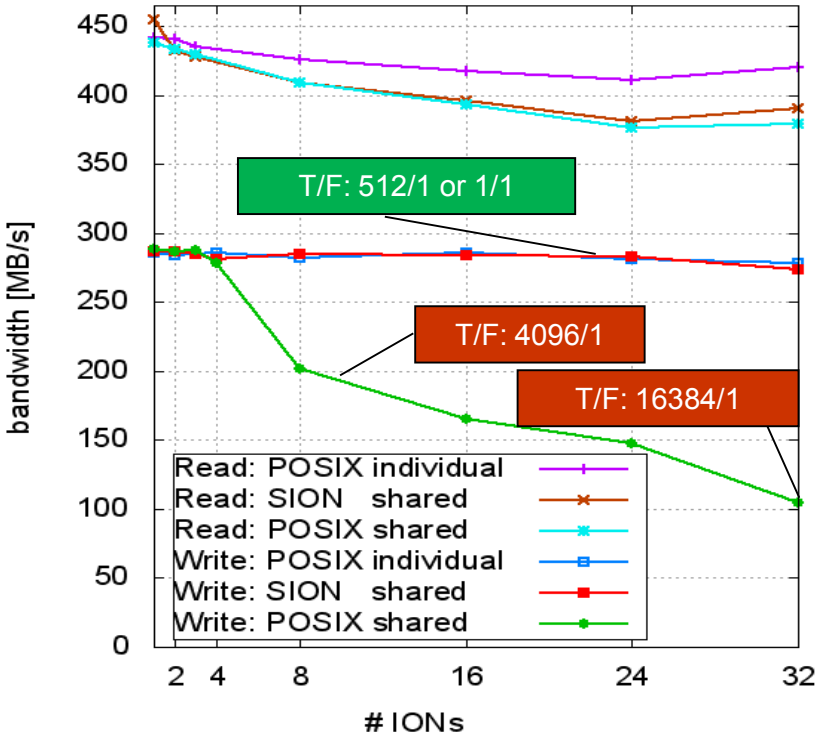
Common I/O strategies

Pitfall 4 – Number of Tasks per Shared File

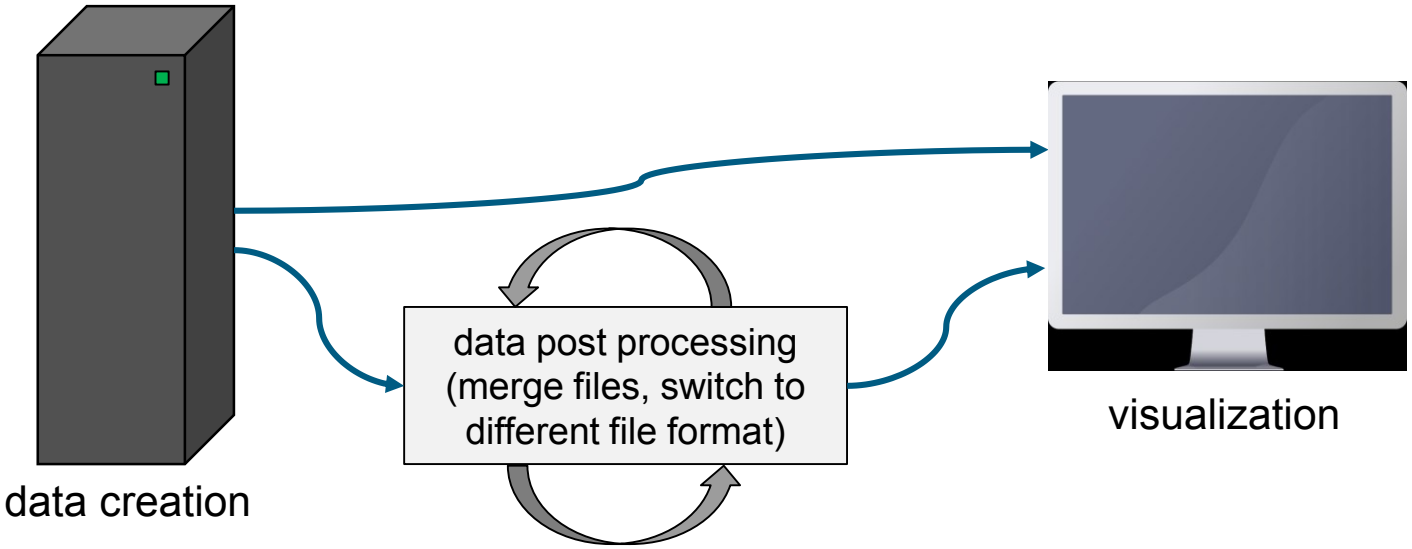
- Meta-data wall on file level
  - File meta-data management
  - Locking



- Example Blue Gene/P
  - Jugene (72 racks)
  - I/O forwarding nodes (ION)
  - GPFS client on ION
  - Solution:
    - → tasks : files ratio ~ const
    - → one file per ION
      implicit task-to-file mapping



I/O Workflow



- Post processing can be very time-consuming (> data creation)
  - Widely used portable data format avoid post processing
- Data transportation time can be long: 1TB $\xrightarrow{1000\ \text{Mbit/s}}$ 2h
  - Use shared file system for file access, avoid raw data transport
  - Avoid renaming/moving of big files (can block backup)

I/O Workflow
Pitfall 5 – Portability

- Endianess (byte order) of binary data
- Example (32 bit):

2.712.847.316

=

10100001 10110010 11000011 11010100

Address	Little Endian	Big Endian
1000	11010100	10100001
1001	11000011	10110010
1002	10110010	11000011
1003	10100001	11010100

- Conversion of files might be necessary and expensive

I/O Workflow
Pitfall 5 – Portability

- Programming language depending memory order

1	2	3
4	5	6
7	8	9

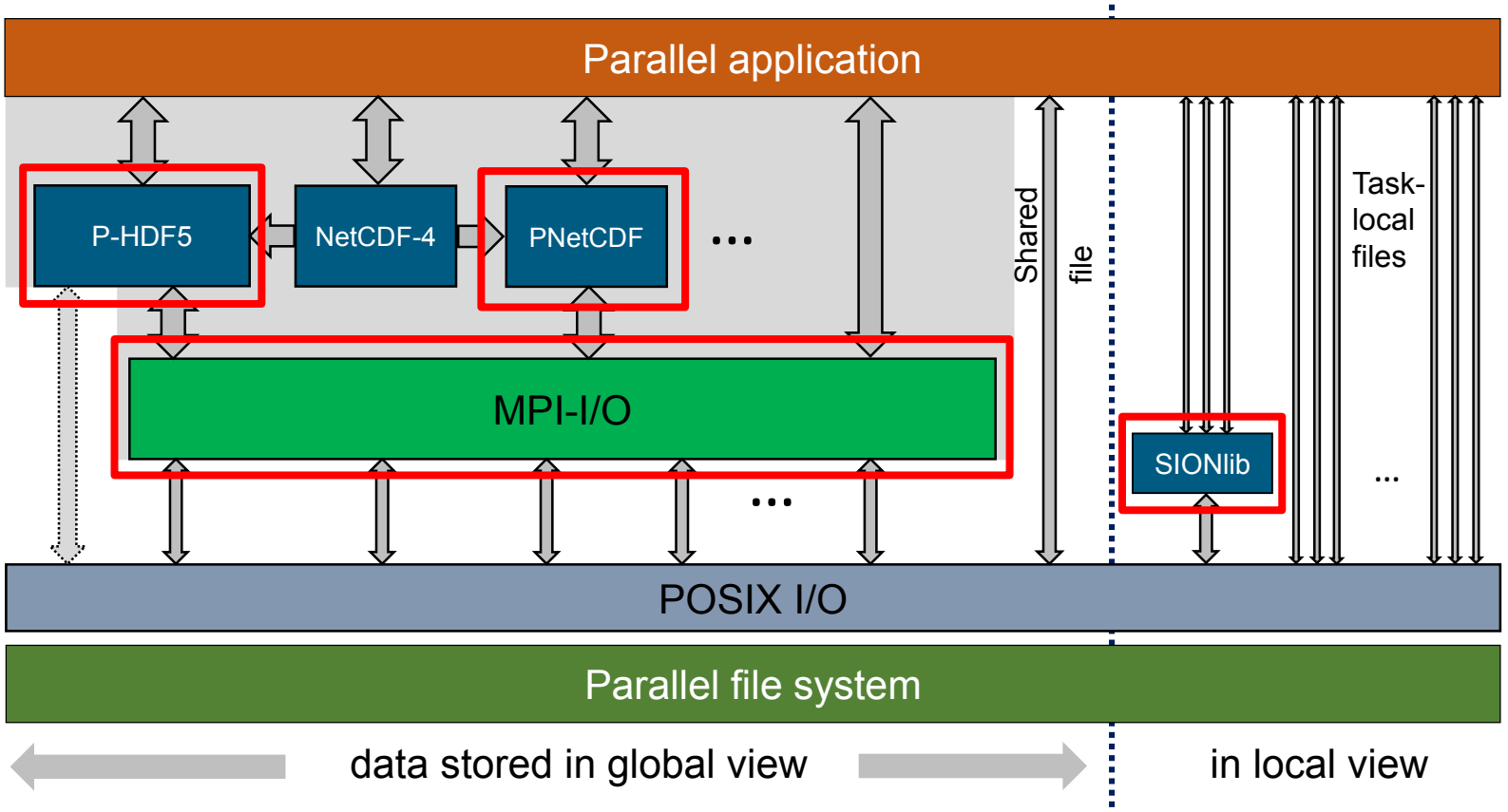
Address	row-major order (e.g. C)	column-major order (e.g. Fortran)
1000	1	1
1001	2	4
1002	3	7
1003	4	2
1004	5	5
…	…	…

- Transpose of array might be necessary when using different programming languages in the same workflow
- Solution: Choosing a portable data format (HDF5, NetCDF)

How to choose an I/O strategy?

- Performance considerations
  - Amount of data
  - Frequency of reading/writing
  - Scalability
- Portability
  - Different HPC architectures
  - Data exchange with others
  - Long-term storage
- E.g. use two formats and converters:
  - Internal: Write/read data “as-is”
    → Restart/checkpoint files
  - External: Write/read data in non-decomposed format
    (portable, system-independent, self-describing)
    → Workflows, Pre-, Post-processing, Data exchange, …

Parallel I/O Software Stack



General exercise workflow

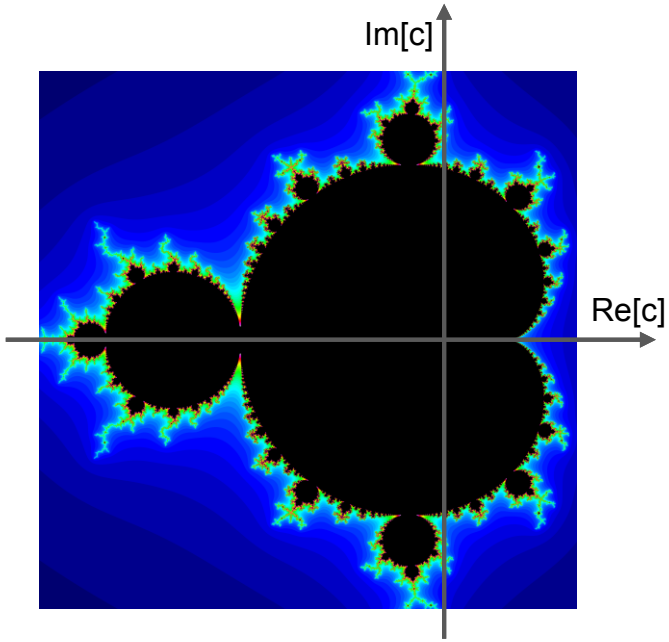
1. Login to your workstation using trainXXX and the given password
2. SSH passphrase is the same password
3. Start a terminal session and login to the JUROPA Cluster:
   `ssh -X juropa`
4. Use `emacs` or `vim` directly on the JUROPA system to avoid copying exercise files from the workstation system
5. Open another terminal window also connected to the JUROPA system
6. Start an interactive computing session:
   `msub -I -X -l nodes=1:ppn=8,walltime=06:00:00`
7. Use the interactive session to execute jobs (using `mpiexec`) and your first session to manipulate your files

Course exercise: Mandelbrot set

Set of all complex numbers c in the complex plane for which

$$z_{n+1} = {z_n}^2 + c$$
$$z_0 = 0$$

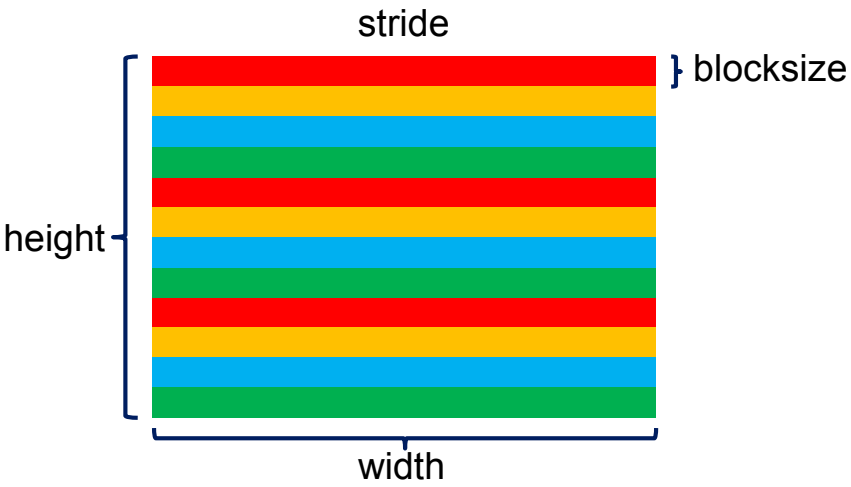
does not approach infinity

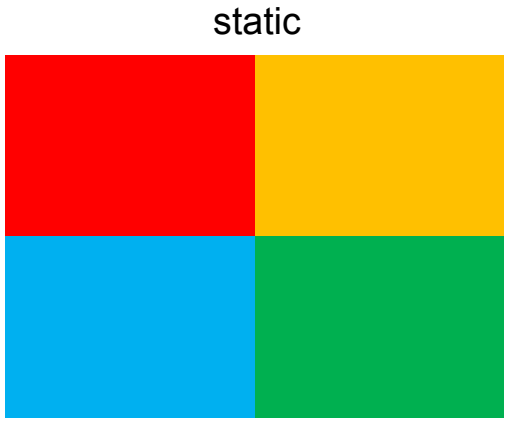


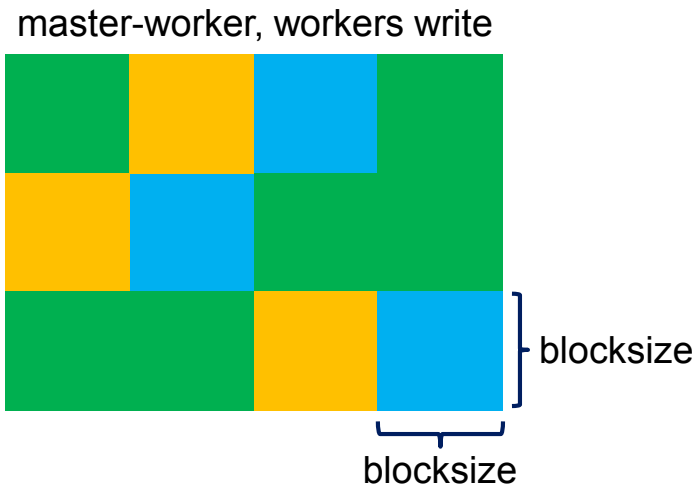
Course exercise: Mandelbrot set

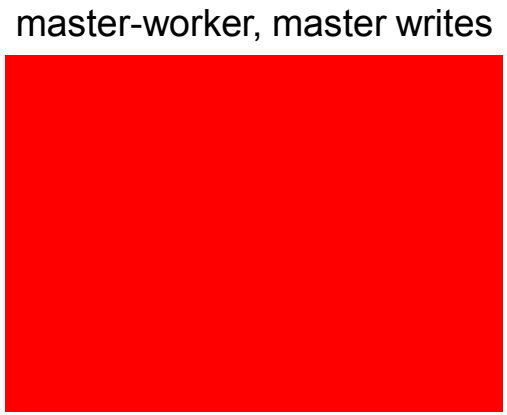
- I/O comparison example
- Four different decomposition types
  - stride
  - static
  - master-worker (workers write)
  - master-worker (master writes)
- Four different output formats
  - parallel-netcdf
  - HDF5
  - Sionlib
  - MPI-IO
- Two different programs
  - `mandelmpi`: parallel Mandelbrot calculation
  - `mandelseq`: serial output picture generation

Decomposition types









mandelmpi

JÜLICH FORSCHUNGSZENTRUM

Command line options

`-v`	use verbose mode
`-t`	decomposition type (`0`: stride, `1`: static, `2`: master-worker master write, `3`: master-worker worker write), default: `0`
`-w`	width, default: `256`
`-h`	height, default: `256`
`-b`	blocksize (not used for type = `1`), default: `64`
`-p`	number of procs in x-direction (only used for type = `1`)
`-q`	number of procs in y-direction (only used for type = `1`)
`-x`	coordinates of initial area: `x1 x2 y1 y2`, default: `-1.5 0.5 -1.0 1.0`
`-i`	max. iterations, default `256`
`-f`	output type (`0`: SIONlib, `1`: HDF5, `2`: MPI-IO, `3`: pnetcdf), default: `0`

mandelmpi

```
mpiexec -np 4 ./mandelmpi -v -t 2 -w 4096 -h 4096 -x -.59 -.54 -.58 -.53 -i 1024 -f 0
```

Example output

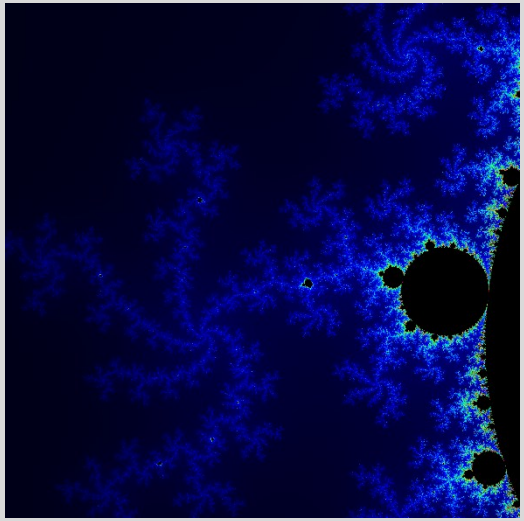
```
using SIONlib
start calculation (x= -0.59 .. -0.54,y= -0.58 .. -0.53)
calc_master[00]: 4096x4096
calc_worker[01]: 64x64
calc_worker[02]: 64x64
calc_worker[03]: 64x64

PE 00 of 04: t= 2 4096 x 4096 bs= 64 calc=   50.859, wait= 26.326,
        io= 716.462, mpi= 7893.249, runtime= 8687.163 (ms)

PE 01 of 04: t= 2 4096 x 4096 bs= 64 calc= 5749.752, wait= 25.301,
        io= 805.705, mpi= 2047.276, runtime= 8688.862 (ms)

PE 02 of 04: t= 2 4096 x 4096 bs= 64 calc= 2651.758, wait= 28.214,
        io= 744.241, mpi= 5258.484, runtime= 8693.876 (ms)

PE 03 of 04: t= 2 4096 x 4096 bs= 64 calc= 4631.728, wait= 42.970,
        io= 793.272, mpi= 3196.786, runtime= 8695.410 (ms)
```

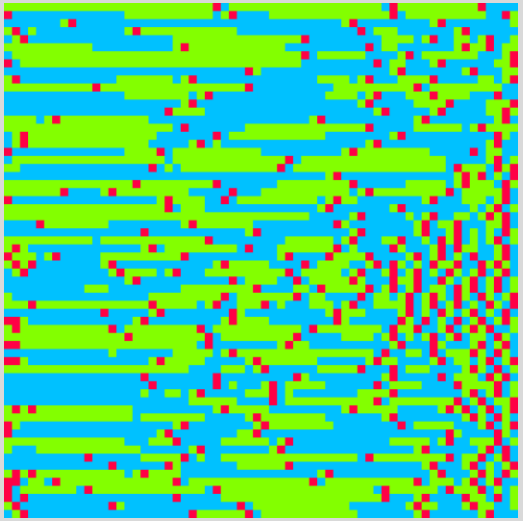
mandelseq

Command line options

`-f` output type (`0`: SIONlib, `1`: HDF5, `2`: MPI-IO, `3`: pnetcdf), default: `0`

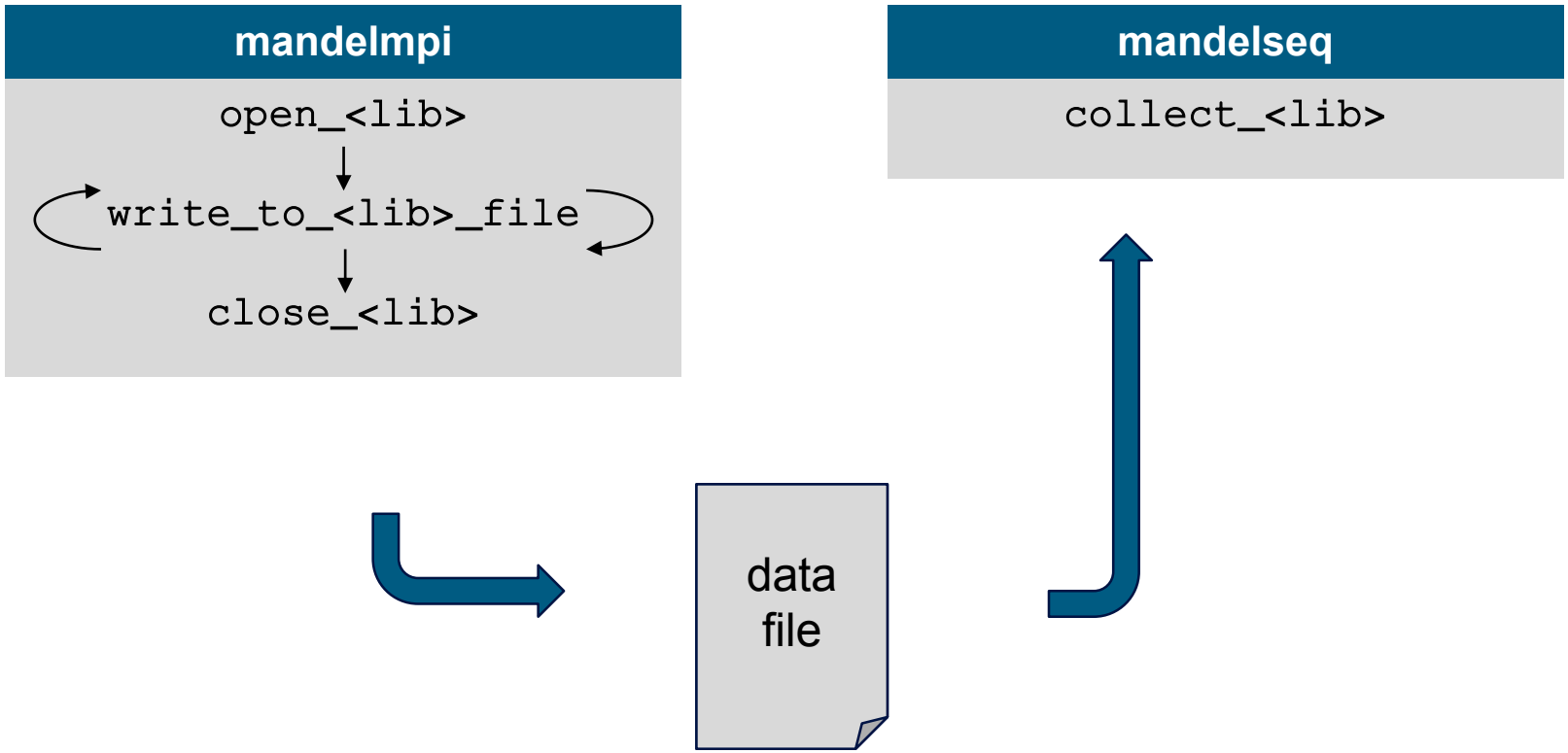
Output





process distribution image
only available for SIONlib

Mandelbrot exercise workflow



Mandelbrot exercise workflow

1. Load modules
   `. load_modules_juropa.sh`
2. Run compilation
   `make`
3. Change runtime parameter in "run.job" file or use `mpiexec` directly in your interactive session
4. Submit job if not running interactive session
   `msub run.job`
5. Create result image
   `./mandelseq -f <format>`
6. View image (**not** in interactive session)
   `display mandelcol.ppm`

Mandelbrot exercise API

Usable info structure

C

```
typedef struct _infostruct
{
  int type; int width; int height;
  int numprocs;
  double xmin; double xmax; double ymin; double ymax;
  int maxiter;
} _infostruct;
```

Fortran

```
type :: t_infostruct
  integer :: type, width, height
  integer :: numprocs
  real :: xmin, xmax, ymin, ymax
  integer :: maxiter
end type t_infostruct
```

Mandelbrot exercise API

Open file in parallel (mandelmpi)

C

```
void open_<lib>(<type> *fid, _infostruct *infostruct,
                int *blocksize, int *start, int rank)
```

Fortran

```
open_<lib>(fid, info, blocksize, start, rank)
  <type>, intent(out) :: fid
  type(t_infostruct), intent(in) :: info
  integer, dimension(2), intent(in) :: blocksize
  integer, dimension(2), intent(in) :: start
  integer, intent(in) :: rank
```

`fid`	lib specific file_id (can occurs twice if multiple ids needed)
`info`	global information structure
`blocksize`	chosen (or calculated) blocksizes (C: [y,x], Fortran: [x,y])
`start`	calculated start point (C: [y,x], Fortran: [x,y], starting at `0`)
`rank`	process MPI rank

Mandelbrot exercise API

Close file in parallel (mandelmpi)

C

```
void close_<lib>(<type> *fid, _infostruct *infostruct,
                 int rank)
```

Fortran

```
close_<lib>(fid, info, rank)
  <type>, intent(inout) :: fid
  type(t_infostruct), intent(in) :: info
  integer, intent(in) :: rank
```

`fid`	lib specific file_id (can occurs twice if multiple ids needed)
`info`	global information structure
`rank`	process MPI rank

Mandelbrot exercise API

Write to file in parallel (mandelmpi)

C

```
void write_to_<lib>_file(
  <type> *fid, _infostruct *infostruct, int *iterations,
  int width, int height, int xpos, int ypos)
```

Fortran

```
write_to_<lib>_file(fid, info, iterations, width, height,
                    xpos, ypos)
  <type>, intent(in) :: fid
  type(t_infostruct), intent(in) :: info
  integer, dimension(:), intent(in) :: iterations
  integer, intent(in) :: width
  integer, intent(in) :: height
  integer, intent(in) :: xpos
  integer, intent(in) :: ypos
```

`iterations`	data array
`width, height`	size of current data block (pixel coordinates)
`xpos, ypos`	position of current data block (pixel coordinates starting at `0`)

Mandelbrot exercise API

Collect parallel written data in serial program (mandelseq)

C

```
void collect_<lib>(
  int **iterations, int **proc_distribution,
  _infostruct *infostruct)
```

Fortran

```
collect_<lib>(iterations, proc_distribution, info)
  integer, dimension(:), pointer :: iterations
  integer, dimension(:), pointer :: proc_distribution
  type(t_infostruct), intent(inout) :: info
```

`iterations`	data array
`proc_distribution`	process distribution array (only in Sionlib)
`info`	global information structure